Membrane-based design and management methodology for parallel dynamically reconfigurable embedded systems

Pamela Wattebled, Jean-Philippe Diguet, Jean-Luc Dekeyser

Membrane-based design and management methodology for parallel dynamically reconfigurable embedded systems

Pamela Wattebled
Universite de Bretagne Sud and
INRIA Lille-Nord Europe
Lorient, France
Email: pamela.wattebled-meftali@univ-ubs.fr

Jean-Philippe Diguet
Universite de Bretagne Sud,
Lab-STICC - CNRS
Lorient, France
Email:jean-philippe.diguet@univ-ubs.fr

Jean-Luc Dekeyser
Universite des Sciences et Technologies
LILF-CNRS
Lille, France
Email:jean-luc.dekeyser@lifl.fr

***Abstract*—Partial and dynamic reconfiguration provides a relevant new dimension to design efficient parallel embedded systems. However, due to the encasing complexity of such systems, ensuring the consistency and parallelism management at runtime is still a key challenge. So architecture models and design methodology are required to allow for efficient component reuse and hardware reconfiguration management.This paper presents a distributed persistence management model and its implementation for reconfigurable multiprocessor systems on dynamically reconfigurable circuits. The proposed approach is inspired from the well-known component based models used in software applications development. Our model is based on membranes wrapping the systems components. The objective is to improve design productivity and ensure consistency by managing context switching and storage using modular distributed hardware controllers. These membranes are distributed and optimized with the aim to design self-adaptive systems by allowing dynamic changes in parallelism degree and contexts migration. Simulation and synthesis results are given to show performances and effectiveness of our methodology.**

I. Introduction

Standards and applications algorithms are changing or evolving more and more frequently, in many domains. This everlasting evolution combined to the important cost of prototyping and designing embedded systems, leads engineers to adaptable and flexible hardware solutions. Thus, FPGAs (Field Programmable Gate Array) emerge as the low cost flexible solution, especially with the significant increase of logic amount that they integrate.

Unlike processors, FPGAs are truly parallel in nature, so that several different processing operations are not competing for the use of resources. Numerous independent processing tasks can be assigned to a specific region of the circuit, and can run independently without depending on any other logic blocks. However, designing FPGA-based systems is a complicate tasks for designers, since no underlining architectural model nor standard hardware and software interfaces are given. So, flexibility and components reuse in such systems are of prime importance in any efficient design methodology targeting FPGAs.

Modern FPGAs support partial and dynamic reconfiguration, and this makes them able to integrate bulky applications. However, important time overhead due to the consistency management and context switch storage/control prevent the full benefit of this feature. Indeed, replacing partial bitstream by another at runtime is generally a complex task. So the system must integrate a mechanism able to manage efficiently the context switch for each reconfiguration. Such a system has to ensure the consistency, the storage and restoration of all relevant information before revoking (respectively loading) any partial bitstream. This control mechanism has also to be efficient by minimizing overhead times, and finally to be modular and flexible, so that it can be generated automatically. The objective of this paper is to present the architectural part of complete methodology developed in the FAMOUS project in order to improve productivity (code reuse) by allowing modular modeling of embedded systems for an implementation on partial and dynamically reconfigurable FPGAs. This whole project includes MDE(Model Driven Engineering methodology)-based methodology for specification and code generation, and configuration controller synthesis but in the paper we only detail our concept for the design of HW components managing parallelism and dynamic reconfiguration. It allows the modeling of both IPs composing the system and the logic ensuring an efficient consistence and context storage/switch at runtime. This functionality is achieved by a distributed hardware mechanism in order to minimize communications and time overhead during the reconfiguration phases. Our approach is dedicated to data-flow applications such as Kahn Process Networks (KPN), without strong real time tasks. The rest of this paper is organized as follow. The next section presents a survey of some significant works on persistence and context switch. Section III introduces the component-based approach and some other necessary concepts of this paper. Our persistence and context switch methodology is detailed in section IV. Section V gives some experiment results and section VI concludes this paper as gives some perspectives.

II. Related work

Many different approaches and methods tried to address the difficult problem of context control and management in reconfigurable systems. These work may be classified into three main categories, depending on the implementation choices. Actually, one can distinguish: hardware, operating system (OS) services and software tasks for the context controller. Hereafter is given a non exhaustive survey of these different approaches.

A. Software based switch control and management

Several researches have been conducted on the software based reconfiguration. In [1] is proposed an architectural style for real-time systems, in which the dynamic reconfiguration is implemented by a synchronous control task in response to the condition changes. To handle runtime dependencies between components, [2] developed a reconfiguration control based on a software task using runtime analysis of interactions to selectively determine which interactions should be allowed to proceed. In [3] a hierarchically self adaptation software models for the robot system to provide fault tolerance has been presented. [4] proposes a service-oriented software framework to handle some design options for dynamic reconfiguration. [5] proposes a programming paradigm using domain-specific elemental units to provide specific guidelines to control engineers for creating and integrating software components by using port-based object.

These work propose in general flexible models, in the sense that they do not take the FPGA architecture into account. However, they present limited performances, especially in terms of execution time. They are also centralized approaches implying important communication overhead.

B. Operating systems

The task of the OS is to support the development and execution of applications on given architectures. Thus, in the context of dynamically reconfigurable FPGAs, the OS may manage context switch and control for both software and hardware tasks. Therefore, such OS are much more complex than conventional ones. [6] presents, in an interesting way, basic issues that can impact the design of an operating system for FPGAs with dynamic reconfiguration capabilities. It proposes that applications be designed into relocatable cores. [7] and [8] discuss high-level OS support for reconfigurable systems. [9] and [10] presented operating systems as intermediate layers between hw and sw. These OS support dynamic reconfiguration and hide hw complexity for the programmer. But, in such systems the reconfiguration support is ensured by single and centralized OS services, so this implies a king of sequentiality when handling reconfigurations.

These approaches lead, unfortunately, to important time overhead and do not really take advantage from huge amount of available logic in recent FPGAs.

C. Hardware

Several research efforts have been dedicated to hardware context switch management. The first work in this topic where dedicated to classical FPGAs (not dynamically reconfigurable) and does not focus on reconfigurability, because they have been used mainly for emulation and prototyping purposes. More recently, this has started to change and new architectures, dedicated for modern dynamically reconfigurable FPGAs have been proposed. Thus for instance [11] proposes architectures able to store few fixed number of contexts to configure LUTs. With this approach, context switch is controlled and implemented in a hardware, it results in the reduction of time overhead, but it treats very low level FPGA architectures. This make it very platform dependent and not flexible or reusable. Reconfigurable computing and FPGAs architectures have been used in many cases for the implementation of control applications. In [12], [13] FPGA implementations of reconfiguration controllers are given. In [14], [15] they used fuzzy logic to try to optimize the switch controller implementations . However, none of them is dynamically reconfigurable. [16] proposes a reconfigurable architecture is using an external memory to store bitstreams and contexts, but the approach is highly architecture dependent and seems time consuming due to the fact that everything in stored in an off-chip memory.

In [17] and [18] authors propose a hardware context switch management mechanism based on used defined registers. The approach relies on designer experiences and thus error prone. It is also not generic and application dependent.

More recently, [19] proposes a wrapper for dynamic context management. It contains simple buffer that sends a context when needed. The main disadvantage of this approach is the dynamicity of the wrappers. This makes the bistreams more complex and may cause important reconfiguration overhead.

III. Software component based approach

In the software world, self-adaptive applications modify their behavior dynamically and autonomously using introspection, recomposition, adding and removing components, in order to adapt to changes that may occur in their execution environment, the reader can refer for instance to [20].

A. The evidence of component model

A component is seen as a black box that can be changed and replaced by another black box having the same interfaces. This is exactly what we want to do in the hardware world.

After the transition, of software designers, from task-based models to the object-based ones [21], moving to component-based models seems to be an obvious solution for the future. In fact, such models allow for the abstraction level and the granularity we are expecting.

B. Business and technical aspects separation

A component is divided into three parts. The first one is the business part of the component, the second is the technical part containing various controls and other services that may be useful to the component, and the third part contains interfaces allowing communication with other components. See Fig. 1. The component based approach ensures a clear separation between the technical (control) part and the business one.

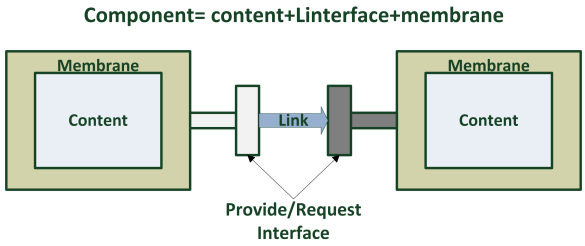

Fig. 1. Component model

This constitutes the majors advantage of such models for their application to our context.

IV. Persistence and context switch methodology

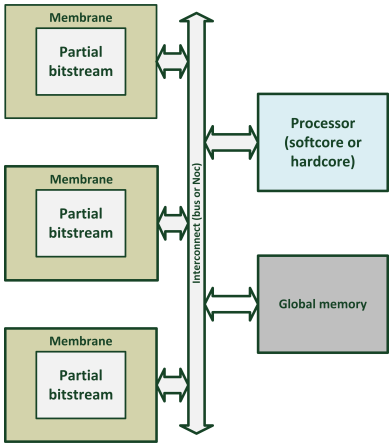

Fig. 2. Membrane-based reconfigurable architecture

Our contribution is a distributed hw support for partial and dynamic reconfiguration as well as communication. This support handles context switching and storage while reconfiguring a system at runtime.
The proposed model is flexible (can be reused for different applications), modular and generic. It is inspired from, the well-known, software component-based models. It is implemented as a hardware membrane that connects IPs with the rest of reconfigurable system (cf Fig. 2).

A. The membrane

The dynamic reconfiguration support consists in a generic membrane wrapping a partial bitstream. Each membrane is attached to one partial bitstream at a given time. However, as a partial bitstream is likely to be replaced by another, regular context backups of the concerned task are performed. These backup operations may enable the tasks to be restored at any time in any membrane. The granularity of the backup is chosen by the hardware designer, this is an important point of an new way of a reconfiguration-aware design methodology, see section IV-G.
Thus the membrane is composed of the backup and data context change services. Therefore, it has a cache memory

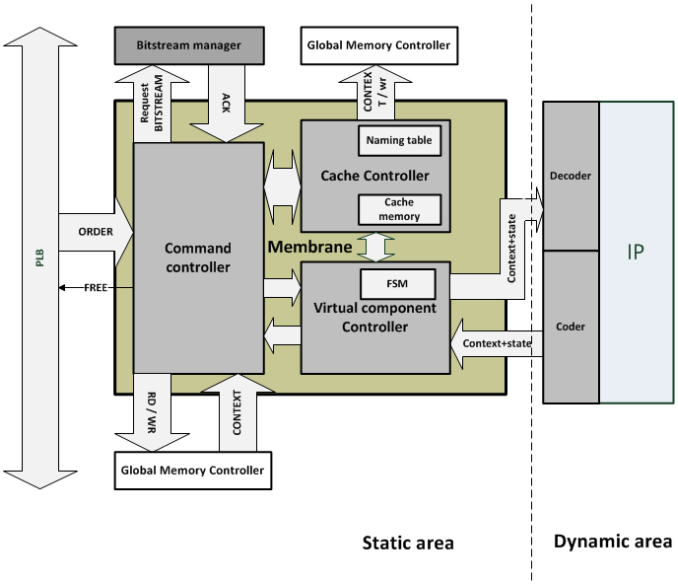

Fig. 3. Membranes architecture and interconnection

that contains context data corresponding to different partial bitstreams as shown in Fig.3. The data contexts are just considered by the membrane as a raw succession of bits. This gives a generic aspect to contexts in the proposed model and therefore a context can be used by different partial bitstreams. Such a property may offer interesting advantages in the case, for example, of changing a task ”A” by anther one ”B” having the same functionality but less energy consuming. In fact, in that case, the stored context of the first task can be restored and used by the second one. This permits ’B’ to continue the iteration where ’A’ was stopped. In the rest of this section, different elements composing the membrane are presented with details Fig.3.

B. Virtual component

While in software, a membrane is fully owned by a single component, our membrane is static and may be attached to different components. However, in the software world, a membrane contains all of the component services and the components itself contains the business part. We keep this property in our work.
In our model we include the principle of virtual component, which is composed of a membrane and a content. Each membrane can support different types of components and each component can be attached to any membrane. However, a given membrane can not be attached to more than one component at the same time. On the other hand, each component belongs only to a single membrane. This results in the principle of virtual component.
The membrane is responsible of controlling dynamic reconfiguration, context save and communications. Its content which is our IP (bitstream) is a black box and contains the business part. The content is implemented on the dynamic part of the FPGA is no longer on the static part where the membrane is

located.
Each component must have checks points, ie breakpoints where relevant are saved within a context. The important property is that our IP can resume execution from the saved context.

C. Membrane input commands

There are two ways to communicate reconfiguration commands to membranes. The first one consists in directly sending a new configuration order. The membrane is then informed that it must execute a given task with a specific context, at a given time. In this case, the membrane will check if this task is already running and if it has the right context. Otherwise, it will either send a request to Bitstream manager. if it the task is missing, or performs a memory read if the context is missing. The second way is used when the running application knows membranes states, in this case it sends the correct order directly to the membrane. The order may be one of the following:

- Change current context (with keeping the current IP)
- Change an IP by another (with keeping the current context)
- Change an IP (with a new context)
- Stop

D. Cache memory

The membrane model integrates a small cache memory able to store a parametrable number of contexts. When this memory is full, contexts are moved to the global memory. This mechanism increases performances, significantly, in terms of reconfiguration time overhead (dependently on the application). The cache is full associative with LRU replacement strategy and has two lines. This choice is made to load a context while the current context continues to be updated.
The length of a line of memory depends on the needs of the application. It should be the maximum size of the biggest context. In the case that contexts have important sizes (e.g. Image processing), the designer can define a context which is composed of variables ID and stored data addresses. In that case the context content is stored in the global memory to avoid bulky local memories in the membranes.

E. Virtual component FSM

The proposed model contains a finite state machine (FSM) with 9 states detailed in Fig.4. This FSM is integrated in the membrane. Transitions from one of its states to another are ensured by different trigger events. As shown in Fig.4, we can distinguish three triggers. The first one is the configuration controller; it is driven by events coming from a remote configuration controller. The second trigger is the IP itself, its events are the ones coming from the reconfigurable hw task, thus the IP must be compliant with the model. The third one is the membrane itself. Fig. 4 shows also the different messages sent by the triggers. For example, to go from 'init' state to 'run/backup' one, the event is triggered by the IP by sending the message 'backup'.
The FSM is initialized when the membrane in instantiated, while the IP is loaded.
When the FSM is in the 'Run /backup' or 'backup context' states, this means that the context sent by the IP is being saved. At any time the FSM may receive the order to stop current execution, either as a strong stop or a stop signal coming from the external controller (configuration Controller). In this case it moves to either wait state consequently to a strong stop or to the backup context state if the received message is a simple stop. This last case is not an emergency stop, so a context store can be realized before moving to the wait mode.
One of the main advantages of this FSM is its genericity, as it is common to all components. We will explain in Section IV-G what IPs designers must integrate while following the design flow, in order to make the IPs able to interact with the FSM.

F. Controllers

The implemented controllers are composite. Thus the cache controller contains the cache memory and the naming table. The virtual component FSM controller includes the FSM and the current status register of the IP.

1) Command controller: After receiving an order from the external configuration controller developed in the FAMOUS project, the command controller asks the cache controller to check if involved contexts are present in the cache. In case of cache miss, it will request it to the cache repository. It also verifies that the correct bitstream is loaded, otherwise it asks the Bitstream manager to load them. Then it transmits remote orders to the FSM controller (IP controller).

2) Virtual component controller: The virtual component controller contains the virtual component's FSM and its current status register. This register is updated by the controller according to the orders received from the command controller. If no order is received from the command controller, it will check if the IP has sent a backup of end job messages.
The possible external orders that the controller can receive are the following :

- INIT: initialization of the context. The message contains the message code, the context name and context itself. The controller updates the status register.
- STOP: stop the IP. The message contains just the message code. The controller sets state register to 'WAIT'.
- RUNBACKUP: restore context. The message contains the message code and the context name. Thee state register is updated by the controller and this later moves to 'BACKUP' state.
- READ: reading a context from the cache. The message contains the message code and the context name. The FSM controller will ask the cache controller for the data.
- READ/DISABLE: reading a context from the cache and disabling it. The message contains the message code and the context name. The FSM controller will ask the cache controller for data and also requests to disable the context.

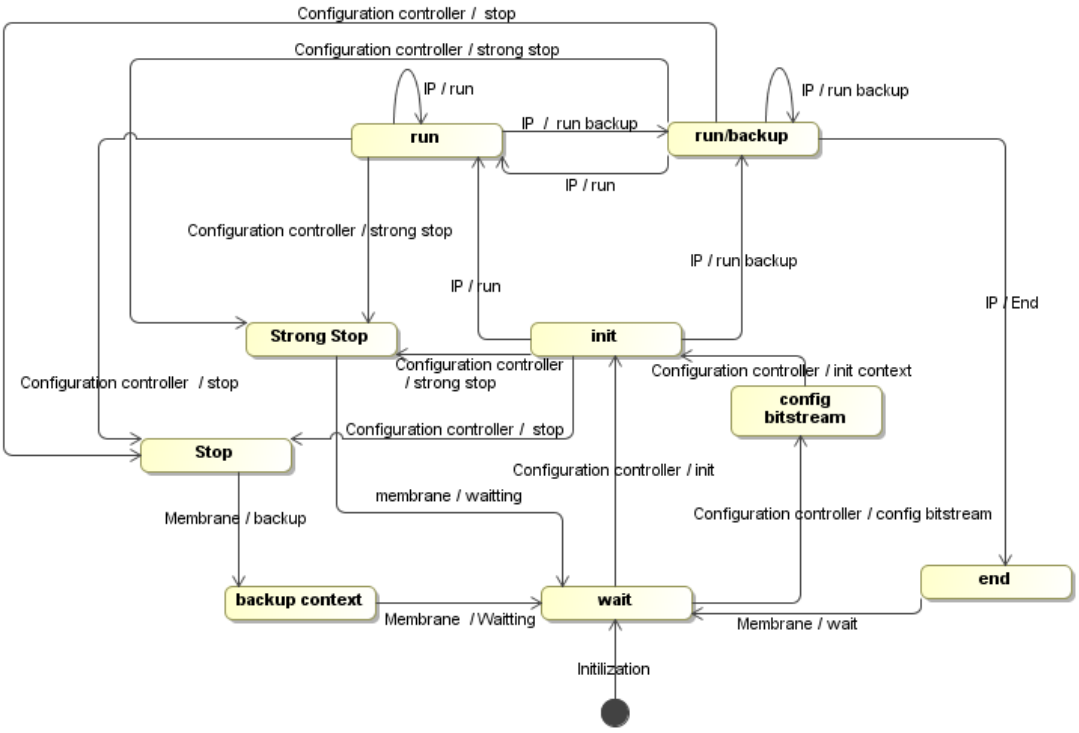

Fig. 4. UML diagram of a virtual component FSM

- DISABLE: disables a context (cache line no longer valid). The message contains message code and the context name. The FSM controller will ask the cache controller to disable the context.

3) Cache controller: The cache controller controls the small memory and naming table in the membrane. It handles 3 orders : write, read and disable a cache line. Two forms of writing as possible. Either the data is not in the cache, then it must be loaded from the global memory, or the data is already in the cache and it consists then in an update. Thus, in order to write, in the cache, we must provide the name of the context, the context itself and set the write bit to 1. To read a context , the name of the context must be given and the read bit 1.

G. Design flow for hardware designs

The designer effort is considerably reduced with the proposed membrane, particularly through the generic aspect of the finite state machine. However, the designer is still in charge of two main tasks. The first one is to design or modifying an existing IP so that it will be able to send its current state. This is also a way to introduce switch points [22] within the IP design, these points are application dependent and can be chosen according to a tradeoff between the amount of data to be stored and the restoration time. The second task is to ensure communication between the IP and the membrane, it means design the coder/decoder according to the membrane standard I/O.

1) IP State: Some transitions between the FSM states are due to events coming from the IP (see Fig.4). Such events must be integrated in the IP's FSM by the designer. We distinguish three different events of this nature:

- Run: When the membrane receives the event 'Run' from the IP, its FSM moves to the 'RUN' state'. This event can occurs when the FSM is in 'RUN', 'INIT' or 'RUN/BACKUP' states.
- Run backup: As for the 'RUN' event, when the membrane receives the event 'RUN/BACKUP' from the IP, its FSM moves to the state 'RUN/BACKUP'. These events can happen from the same states as the event 'Run'.
- End: Event 'END' occurs when the FSM is in the 'RUN/BACKUP' state, forcing the context save at the IP execution end. This event notifies to the membrane that the execution of a task is completed, so that the membranes FSM goes therefore to the 'WAIT' state.

2) I/O standard: The approach has to be generic, so a context is only seen as a raw succession of bits by the membrane. Thus when a context is sent to the IP, it must be divided into groups of bits to form the real data. On the other side, outputs of the IP must be grouped in order to store the context only as a succession of bits. The designer has then to design a coder and a decoder around its IP.

V. Experiments

In order to show the effectiveness and the easy use of the proposed membrane, we chose a simple application that does not show a big IP design effort, but allows to describe the behavior, the integration and the performances of the membrane. Thus the chosen application consists in a HW IP

computing the factorial of a given integer. The input (n) and output ($n!$) of this IP are 32 bits integers. In this case, the original IP has the three following states:

- Initialization: receiving the input integer 'n'.
- End of an iteration: at the end of the iteration 'i', the product of integers from 'n' to 'i' has been calculated.
- End of task: the IP has completed the computation of the factorial.

Thus, the two context data of this task are the input number 'n' and the 32 bits integer resulting from the end of an iteration. Because such values allows continuing factorial calculation, from the result of the iteration, where the system was stopped. In this case, it is needed to store a context composed by two 32 bits integers. Therefore we need 64 bits to store it.

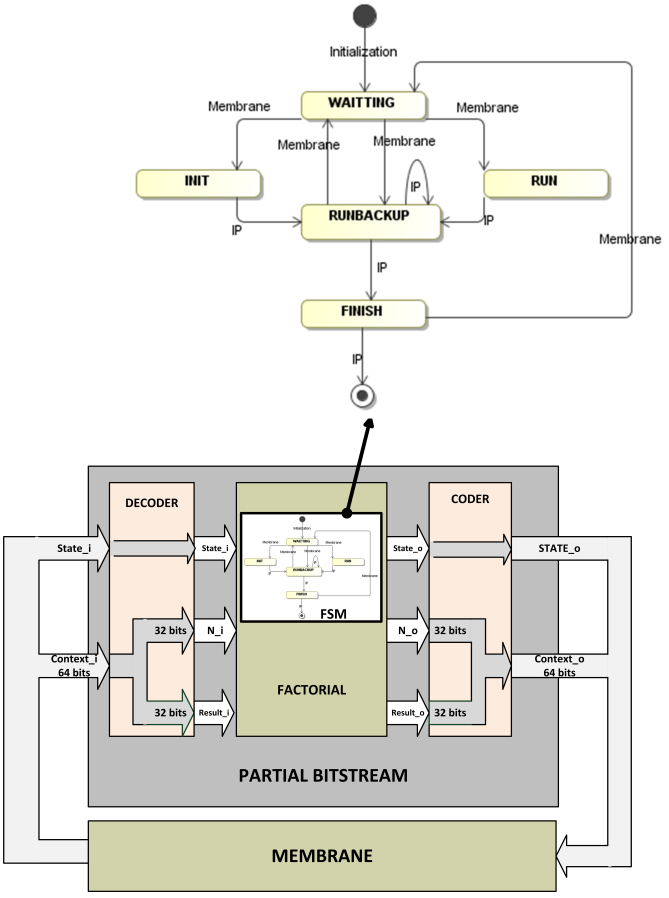

Fig. 5. Membrane-based architecture of reconfigurable factorial application

A. Dynamic reconfiguration support integration

In order to integrate our dynamic reconfiguration support to this application, the IP factorial has been modified by integrating to it three main elements: a single FSM, a context coder and a context decoder. The resulting component takes two inputs and furnishes two outputs instead of one of each in the original IP as shown in Fig.5. The use of these new ports is illustrated later in this section. The decoder takes the 64 bits of context, separates into two, the first 32 bits for the integer and the remaining 32 bits for the intermediate calculations, the coder implement the reverse transformation.
A generic membrane has been also added to the system in order to control and manage contexts storage and switches.

B. Simulation Scenarii

The objective of our simulation is to show the correctness of the proposed approach, in terms of functionality. Several scenarios have been realized, but we chose to present a simple one in order to keep this section clear and easily understandable. Thus, the scenario consists, initially, in calculating the factorial of 9. As shown in the figure 6, the membrane, in INIT state, sends a two parts message, to the IP. 'INIT' to inform the IP that it will receive a new context, and the context itself containing numbers 9 and 1 representing the number for which we want to calculate the factorial and the result of products accumulation at iteration zero respectively. Thus, the IP receives 'INIT', 9 and 1 from the context decoder. It performs the first iteration, moves to the state 'RUN/BACKUP' and sends throw its outputs state_o, N_o and result_o respectively the values 'RUN/BACKUP', 8 and 9. The context coder concatenates N_o and result_o then sends the obtained context (context_o) and the state 'RUN/BACKUP' to the membrane, as represented by the message 4 fig 6. After receiving this message, the membrane stores the context and changes its own state to 'RUN / BACKUP'. The IP continues its execution and the decoder continues to send intermediate results throw output ports of the IP.

After the second iteration, the system needs a high priority computation and asks the IP to calculate factorial of 3. Thus, the membrane stops the current execution by sending the message 'WAITING' to the IP (message 7). Then, the IP moves to the 'WAIT' state and stops executing. Similarly to the message (1), the message 9 initializes a new context to execute and the IP starts its execution to calculate factorial 3. After two iterations the execution is completed, the results and the state 'FINISH' are sent by the coder to the membrane (message 14). After receiving the message "FINISH', the membranes FSM moves to the state 'WAIT'. The calculation of factorial 9 can then be resumed. In the message 15, we resume its computation with 72 as intermediate result and factorial 7 remains to calculate. The execution continues normally.

C. Synthesis results and analysis

Our simulations and synthesis have been realized on a Xilinx Virtex 6 FPGAs integrated in a ML605 card. As shown in table 1, the membrane uses very limited amount of the available resources. If the local is excluded, the membrane requires 794, 437 and 26 of available register slices, LUT slices and RAM/FIFO blocks respectively. These consumed HW resources are independent from the IP complexity. The overhead due to the cache can be variable since the line sizes and number can be parameterized. In this example two lines of 64 bits are been implemented. Thus, we clearly observe that

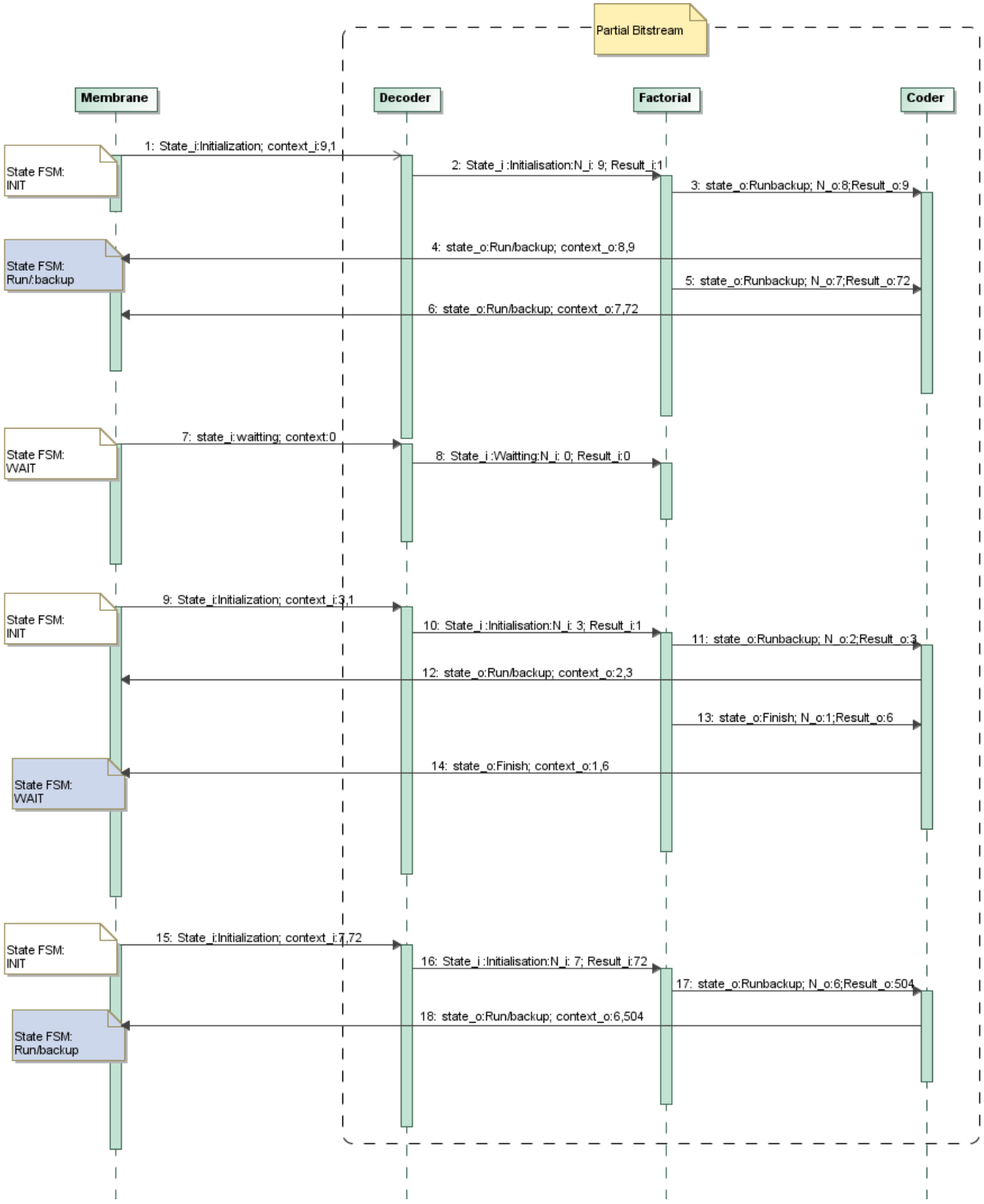

Fig. 6. UML diagram of a simulation scenario

hw overhead due to a membrane implementation is not really significant on a Virtex 6 device and more than 10 times less important on recent device as a Virtex 7T [23]. This allows us to imagine highly parallel reconfigurable architectures using our membranes based approach.

As shown in table 2, a membrane treats the four different possible orders, coming from the systems reconfiguration controller (Microblaze for instance) in a fast way, going from 3 cycles for a STOP to 14 cycles for RUN IP + CONTEXT with a miss and a full cache (with a 100 Mhz clock frequency).

	Membrane with cache memory		Membrane without cache memory	
Number of slice register	794	0,63%	794	0,63%
Number of slice Luts	536	0,35%	437	0,28 %
Number of of block RAM/FIFO	26	6%	26	6%

TABLE I
SYNTHESIS RESULTS/VIRTEX6 XC6VLX240T

	cache hit	cache miss	cache miss and cache full
Run context	8	11 GMA	12 GMA
Run IP	4 LB	4 LB	4 LB
Run IP + context	10 LB	13 GMA,LB	14 GMA,LB
Stop	3	3	3

TABLE II
NUMBERS OF CYCLES BY ORDER (GMA:GLOBAL MEMORY ACCESS COST, LB:LOAD BITSTREAM COST)

VI. CONCLUSION AND PERSPECTIVES

The major benefits of FPGA-based computing are the ability to execute larger hardware designs with fewer gates and to ensure the flexibility of a software-based solution while retaining the execution speed of a more hardware-centric approach. This capability is ensured mainly by the partial and dynamic reconfiguration features.

In this paper is presented a novel hardware based approach for context control and management for parallel and dynamically reconfigurable systems. The proposed methodology is based on distributed, modular, reusable and flexible membranes. It is inspired from the well-known component based approach in software development permits sharing, storing and changing automatically different contexts of hw IPs without using complex OS services. In addition to that, its overhead in terms of used area in not significant compared to the available logic in recent FPGA devices.

The ongoing work deals now the design of the global context manager. It will contain a global naming table able to locate any context (in membranes caches or in the global memory). Such a manager will make controlling dynamic reconfiguration with a Microblaze and the communication between membranes easier and simpler. In cooperation with partners of the FAMOUS project, the next steps are the automatic generation of the membrane code by means of a MDE methodology and the design of a demonstrator implementing a complex video application relying on a highly parallel architecture.

REFERENCES

[1] V. Gafni, *Robots: a real-time systems architectural style*, in proceeding ESEEC, Toulouse, 1999.

[2] J. Almeida, M. Wegdam, M. Sinderen, and L. Nieuwenhuis, *Transparent Dynamic Reconfigurable for CORBA*, in proceeding of DOA, Sept., 2001, Rome, Italy.

[3] J. Cobleigh et al, *Containment units: a hierarchically composable architecture for adaptive systems*, in proceeding ACM SIGSOFT Software Engineering Notes, 27(6):159-165, November 2002.

[4] B. MacDonald, B. Hsieh, and I. Warren, *Design for Dynamic Reconfiguration for Robot Software*, in proceeding of ICARA, Dec, 2004, Palmerston North, New Zealand.

[5] D. Stewart, R. Volpe, and P. Khosla, *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects*, in proceeding IEEE Trans. On Software Engineering, vol. 23, no. 12, December 1997.

[6] G. Brebner,*A virtual hardware operating system for the Xilinx XC6200* , in the 6th IWFPLA, 1996.

[7] H. El Gindy, M. Middendorf and all, *Task rearrangement on partially reconfigurable FPGAs with restricted buffer* , in fPLA Workshop, 2000.

[8] G. Wigley and D. Kearney, *The development of an operating system for reconfigurable computing*, in IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, 2001.

[9] Y.Eustache and J-Ph.Diguet, *Specification and OS-based implementation of self-adaptive, hardware software embedded systems* , in CODES-ISSS, Atlanta, USA, 2008.

[10] http://users.polytech.unice.fr/ fmuller/fosfor/

[11] M. Yamashina and M. Motomura, *Reconfigurable computing:Its concept and a practical embodiment using newly developed dynamically reconfigurable logic (DRL) LSI* , in 5th ASPDAC, 2000.

[12] Y. F. Chan, M. Moallem, and W. Wang, *Efficient implementation of PID control algorithm using FPGA technology* , in 43rd Conference on Decision and Control, 2004.

[13] W. Zhao, B. H. Kim, A. C. Larson, and R. M. Voyles, *FPGA implementation of closed-loop control system for small-scale robot* , in 12th International Conference on Advanced Robotics, 2005.

[14] V. Tipsuwanporn, T. Runghimmawan and all, *Fuzzy logic PID controller based on FPGA for process control* , in International Symposium on Industrial Electronics, 2004.

[15] P. T. Vuong, A. M. Madni, and J. B. Vuong, *VHDL implementation for a fuzzy logic controller* , in World Automation Congress, 2006.

[16] D. Kim, *An implementation of fuzzy logic controller on the reconfigurable FPGA system* , IEEE Transactions on Industrial Electronics, vol. 47, no. 3, pp. 703 to 715, 2000.

[17] J. C. Ferreira and M. M. Silva *Runtime reconfiguration support for FPGAs with embedded CPUs: The hardware layer*, , in IPDPS, 2005.

[18] J. Torresen and K. Vinger*High Performance Computing by Context Switching Reconfigurable Logic*, , in proc. of ESMMS 2002.

[19] C.-H. Huang, K.-J. Shih et al, *Dynamically swappable hardware design in partially reconfigurable systems*, in Proc. of ISCAS, USA, 2007.

[20] M. Simonot, V.Aponte, *A declarative formal approach to dynamic reconfiguration*, in Proc., of IWOCE'09, NY, USA 2009.

[21] J. Dondo, F. Rinc, et al., *Dynamic reconfiguration management based on a distributed object model*, In Proc. of FPL, Netherlands, 2007.

[22] J-Y. Mignolet, V. Nollet, P. Coene, et al., *Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip*, In Proc. of FPL, in DATE, 2003.

[23] http://www.xilinx.com